# VeriFast: Imperative Programs as Proofs

Bart Jacobs*, Jan Smans, and Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bart.jacobs,jan.smans,frank.piessens}@cs.kuleuven.be

**Abstract.** This paper describes the VeriFast prototype program verification tool, which implements a separation-logic-based approach for the specification and verification of safety properties of pointer-manipulating imperative programs. The approach's distinctive feature is that it combines very good and predictable verification performance with powerful proofs written conveniently as part of the program. We describe the tool's support for the C language.
The paper introduces the tool's various features by means of a running example of a linked list implementation. A detailed formalization of the core of the approach and a soundness proof are available on the website.

## 1 Introduction

VeriFast is a research prototype program verification tool for verification of safety properties of C and Java programs, based on separation logic.

The safety properties to be verified are specified as annotations in the source code, in the form of function preconditions and postconditions expressed as separation logic assertions. To enable rich specifications, the user may include additional annotations that define inductive datatypes, primitive recursive pure functions over these datatypes, and abstract predicates (i.e. named, parameterized assertions). Abstract predicates may be recursive. A restricted form of existential quantification is supported in assertions in the form of pattern matching.

Verification is based on forward symbolic execution, where memory is represented as a separate conjunction of points-to assertions and abstract predicate assertions, and data values are represented as first-order logic terms with a set of constraints. Abstract predicates must be folded and unfolded explicitly using ghost statements. Rewritings of the abstract state that require induction, or derivations of facts over data values that require induction, can be done by defining *lemma functions*, which are like ordinary C functions except that it is checked that they terminate. Specifically, when a lemma function performs a recursive call, either the recursive call must apply to a strict subset of memory, or one of its parameters must be an inductive value whose size decreases at each recursive call.

Assertions over data values are delegated to an SMT solver, formulated as queries against an axiomatization of the inductive datatypes and recursive pure

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

functions. Importantly, no exhaustiveness axioms are included in this axiomatization; this prevents the SMT solver from performing case analysis on inductive values. Combined with a measure to prevent infinite reductions due to *self-feeding recursions*, this ensures termination of the SMT solver.

A prototype implementation, a formalization and soundness proof, a tutorial text, and a large number of example annotated programs are available at http://www.cs.kuleuven.be/~bartj/verifast/. The implementation includes an IDE that enables the user to step through a failed execution trace, inspecting the symbolic state at each step.

## 2 Walkthrough

We introduce the approach through an example annotated C program that implements a linked list ADT. Successive figures show successive fragments of the example program.

### 2.1 Symbolic execution and predicates

VeriFast performs symbolic execution. The symbolic state consists of three parts: the symbolic heap, the symbolic store, and the path condition. The symbolic heap is a bag of *chunks*. A chunk consists of a predicate name and an argument list. Chunk arguments are terms of first-order logic. The predicate name may be the predicate corresponding to a struct field, in which case we call the chunk a *points-to chunk*. A points-to chunk has two arguments: a term denoting the address of the struct, and a term denoting the field value. The predicate name may also refer to a user-defined predicate.

The symbolic store maps local variable names to terms that denote the variable's current value. The path condition is a set of formulae of first-order logic. These constrain the interpretation of the logical symbols used in the terms in the symbolic store and the symbolic heap.

In the implementation, the terms are SMT solver terms, and the path condition corresponds to the state of the SMT solver. The verifier pushes formulae into the SMT solver during symbolic execution, and pops them when a branch of symbolic execution is finished and the next branch is started.

Notice that this means that the SMT solver does not come into contact with the heap. The heap is dealt with syntactically within the verifier itself. This avoids the quantified formulae required by verification condition generation-based approaches to describe heap effects.

Figure 1 shows function *createNode*. It uses an abstract predicate [16] to hide the internal layout of a node. The **close** ghost statement removes the points-to chunks for the individual fields of $n$ from the symbolic heap, and adds a *node* abstract predicate chunk, as expected by the postcondition. The asterisk denotes separating conjunction: $P * Q$ holds if the heap can be split into two separate parts such that $P$ holds for one part and $Q$ for the other. As we will see, operationally, in our tool the asterisk means sequential composition of consumption

```
struct node { struct node *next; int value; };

predicate node(struct node *n, struct node *next, int value) =
  n→next ↦ next * n→value ↦ value * malloc_block_node(n);

struct node *create_node(struct node *next, int value)
  requires emp; ensures node(result, next, value);
{
  struct node *n := malloc(sizeof(struct node));
  n→next := next; n→value := value;
  close node(n, next, value); return n;
}
```

**Fig. 1.** Example demonstrating abstract predicates and ghost statements (Note: annotations are shown on a gray background. Also, for readability, we typeset some operators differently from the implementation.)

or production of assertions. That is, consuming $P * Q$ means first consuming $P$ and then $Q$; producing $P * Q$ means first producing $P$ and then $Q$.

When symbolically executing function *create_node*, the successive symbolic states are as follows:

```
// h = ∅, s = {next ↦ next, value ↦ value}, Σ = ∅
struct node *n := malloc(sizeof(struct node));
// h = {node_next(n, value0), node_value(n, value1), malloc_block_node(n)},
// s = {n ↦ n, next ↦ next, value ↦ value}, Σ = {n ≠ 0}
n→next := next; n→value := value;
// h = {node_next(n, next), node_value(n, value), malloc_block_node(n)},
// s = {n ↦ n, next ↦ next, value ↦ value}, Σ = {n ≠ 0}
close node(n, next, value);
// h = {node(n, next, value)}
// s = {n ↦ n, next ↦ next, value ↦ value}, Σ = {n ≠ 0}
```

Italic names denote program variables; sans serif names denote logical symbols. At the start of symbolic execution, fresh symbols are generated for the function arguments. In the example, fresh symbols are also generated by the *malloc* call to denote *malloc*'s return value and the initial values of the newly allocated fields.

Figure 2 shows a way to denote a piece of memory containing a set of consecutive nodes. Specifically, abstract predicate $lseg(n1, n2, v)$ represents a set of consecutive nodes where the first node is at *n1* and the last node's next pointer points to *n2*, and the nodes store the list of integers $v$. As a special case, if *n1* equals *n2*, the predicate denotes the empty piece of memory.

During symbolic execution of a function, assertions are *produced* and *consumed*. Producing a points-to assertion or an abstract predicate assertion means adding the corresponding chunk to the symbolic heap, and consuming it means

```
inductive list = nil | cons(int, list);
predicate lseg(struct node *n1, struct node *n2, list v) =
  n1 = n2 ? v = nil : node(n1, ?n, ?h) * lseg(n, n2, ?t) * v = cons(h, t);
```

**Fig. 2.** Example demonstrating inductive datatype definitions, recursive abstract predicates, conditional assertions, and pattern matching

removing a matching chunk from the symbolic heap. If no matching chunk is present in the symbolic heap, an error is reported. If the assertion being consumed contains patterns, the matching process binds the pattern variables; their scope includes the rest of the assertion, or if the pattern occurs in a function body or precondition, its scope includes the rest of the function. Producing a pure assertion (i.e., a boolean expression), means adding it to the path condition, and consuming it means asking the SMT solver to check that it follows from the current path condition. Producing or consuming a separate conjunction means first producing, resp. consuming the first operand, and then producing, resp. consuming the second operand. Producing or consuming **emp** does nothing. If during execution of a function, a conditional construct is encountered, then the remainder of the execution is performed once for each branch of the construct, after adding the corresponding constraint to the path condition. The conditional constructs include the if and switch statements, the if-then-else assertions, and the switch assertions.

Execution of a function starts with an empty symbolic heap and an empty path condition. Then, the precondition is produced. Then, each statement is executed. And finally, the postcondition is consumed. If subsequently, any chunks are left in the symbolic heap, this is considered a potential memory leak and an error is reported. Execution of a function call statement proceeds by first consuming the call's precondition and then producing its postcondition. Execution of an open ghost statement proceeds by first consuming the abstract predicate assertion and then producing its body. Execution of a close ghost statement proceeds by first consuming the predicate's body and then adding the abstract predicate chunk. Patterns may be used as abstract predicate arguments in an open statement, but in the current implementation they cannot be used as arguments in a close statement.

Figure 3 shows the first part of the client-visible interface of the linked list ADT. The implementation keeps a sentinel node at the end of the list, and it keeps a pointer to the first node and to this sentinel node. The dummy patterns (_) in the definition of the *llist* predicate indicate that the *next* and *value* fields of the sentinel node are insignificant.

```
struct llist { struct node *first; struct node *last; };

predicate llist(struct llist *l, list v) =
  l→first ↦ ?fn * l→last ↦ ?ln * lseg(fn, ln, v) * node(ln, _, _) * malloc_block_llist(l);

struct llist *create_llist()
  requires emp;
  ensures llist(result, nil);
{
  struct llist *l := malloc(sizeof(struct llist));
  struct node *n := create_node(0, 0); l→first := n; l→last := n;
  close lseg(n, n, nil); close llist(l, nil);
  return l;
}
```

**Fig. 3.** Example demonstrating dummy patterns

```
lemma void distinct_nodes(struct node *n1, struct node *n2)
  requires node(n1, ?n1n, ?n1v) * node(n2, ?n2n, ?n2v);
  ensures node(n1, n1n, n1v) * node(n2, n2n, n2v) * n1 ≠ n2;
{
  open node(n1, _, _); open node(n2, _, _);
  close node(n1, n1n, n1v); close node(n2, n2n, n2v);
}
```

**Fig. 4.** Example demonstrating lemma functions, distinctness constraint production and patterns in open statements

### 2.2 Lemma functions

Figure 4 shows a *lemma function*, which is like a C function except that it is declared in an annotation and the verifier checks that it terminates and that it has no effect on memory (i.e. it does not allocate, free, or write to memory). The only effect of calling a lemma function is that it rewrites the symbolic heap into a semantically equivalent but syntactically different one, and/or that it adds constraints to the path condition.

In this example, there is no net change to the symbolic heap; all the lemma function does is add a constraint. Specifically, given two nodes for which there are separate abstract predicate chunks in the symbolic heap, the lemma produces a constraint that says that the nodes are distinct.

Such distinctness constraints are not produced automatically by the verifier for abstract predicate chunks, since the fact that two abstract predicate chunks referring to the same abstract predicate appear in memory does not imply any-

thing about distinctness of the arguments. However, the verifier produces them for points-to chunks. Specifically, when producing a points-to assertion $t{\rightarrow}f \mapsto v$, then for any existing points-to chunk $t'{\rightarrow}f \mapsto v'$ in the symbolic heap, a constraint $t \neq t'$ is added automatically. In the example, this occurs during execution of the second open statement.

```
fixpoint list add(list v, int x) {
  switch (v) {
    case nil : return cons(x, nil);
    case cons(h, t) :
      return cons(h, add(t, x));
  }
}

lemma add_lemma(struct node *n1,
    struct node *n2, struct node *n3)
  requires lseg(n1, n2, ?v)
    * node(n2, n3, ?x) * node(n3, _, _);
  ensures lseg(n1, n3, add(v, x))
    * node(n3, _, _);
{
  distinct_nodes(n2, n3);
  open lseg(n1, _, _);
  if (n1 = n2) {
    close lseg(n3, n3, nil);
  } else {
    distinct_nodes(n1, n3);
    open node(n1, ?n1n, ?n1v);
    add_lemma(n1n, n2, n3);
    close node(n1, n1n, n1v);
  }
  close lseg(n1, n3, add(v, x));
}
```

```
void add(struct llist *l, int x)
  requires llist(l, ?v);
  ensures llist(l, add(v, x));
{
  open llist(l, v);
  struct node *n := create_node(0, 0);
  struct node *nl := l→last;
  open node(nl, _, _);
  nl→next := n;
  nl→value := x;
  close node(nl, n, x);
  l→last := n;
  struct node *nf := l→first;
  add_lemma(nf, nl, n);
  close llist(l, add(v, x));
}
```

**Fig. 5.** Example demonstrating fixpoint functions and recursive lemma functions

Figure 5 shows the second client-visible list ADT function, function *add*. It adds a value to the end of the list. Its contract describes its effect on the ADT's abstract value using the *fixpoint function add*. (Note that fixpoint function names and non-fixpoint (i.e., regular or lemma) function names are in separate namespaces; the former may occur only in expressions in annotations, whereas the latter may occur only in call statements.)

A fixpoint function is not allowed to read or modify memory. Its body must be a switch statement over one of the function's parameters. We call this parameter the function's *inductive parameter*. The body of each clause of the switch

statement must be a return statement. A fixpoint function may call other fixpoint functions, but not regular functions or lemma functions. Furthermore, to ensure termination, any call must either be a call of a fixpoint function declared earlier in the program, or it must be a direct recursive call where the argument for the inductive parameter is a variable bound by the switch statement.

Regular function *add* creates a new node to serve as the new sentinel node, then updates the old sentinel node's fields, and finally calls the lemma function *add_lemma* to merge the old sentinel node into the *lseg* abstract predicate chunk. The lemma function does so using recursion. Lemma functions may perform recursive calls, but only direct recursive calls, and termination is ensured by checking that at each recursive call either the size of the piece of memory that the function operates on decreases (specifically, after consuming the precondition there must be a points-to chunk left in the symbolic heap), or, similar to a fixpoint function, the function's body is a switch statement over one of its parameters, and the argument for the inductive parameter in the recursive call is bound by this switch statement. Note that a recursive lemma constitutes an inductive proof of the fact that the precondition implies the postcondition.

```
int removeFirst(struct llist *l)
  requires llist(l, ?v) * v ≠ nil; ensures llist(l, ?t) * v = cons(result, t);
{
  open llist(l, v);
  struct node *nf := l→first; open lseg(nf, ?nl, v); open node(nf, _, _);
  struct node *nfn := nf→next; int nfv := nf→value; free(nf); l→first := nfn;
  open lseg(nfn, nl, ?t); close lseg(nfn, nl, t); close llist(l, t);
  return nfv;
}
```

**Fig. 6.** Example demonstrating execution splits due to conditional constructs in the bodies of predicates being opened

### 2.3 Case splits

Figure 6 shows a function that removes the first element from a list. It requires that the list is non-empty. When the *lseg* starting at *nf* is opened, an execution split occurs because the body of this predicate is an if-then-else assertion. The verifier notices immediately that the then branch is infeasible and does not continue execution on this branch.

Notice also that the first node is freed. A statement *free*$(p)$; looks for a chunk of the form *malloc_block_*$T(p)$, and then for points-to chunks of the form $p{\rightarrow}f \mapsto v$, for each field $f$ of $T$, and it removes all of these chunks.

```
void dispose(struct llist *l)
  requires llist(l, _);
  ensures emp;
{
  open llist(l, _);
  struct node *n := l→first;
  struct node *nl := l→last;
  while (n ≠ nl)
    invariant lseg(n, nl, _);
  {
    open lseg(n, nl, _);
    open node(n, _, _);
    struct node *next := n→next;
    free(n);
    n := next;
  }
  open lseg(n, n, _);
  open node(l, _, _);
  free(nl);
  free(l);
}
```

**Fig. 7.** Example demonstrating loops

```
void main()
  requires emp;
  ensures emp;
{
  struct llist *l := create_llist();
  add(l, 10);
  add(l, 20);
  add(l, 30);
  add(l, 40);
  int x0 := removeFirst(l);
  assert(x0 = 10);
  int x1 := removeFirst(l);
  assert(x1 = 20);
  dispose(l);
}
```

**Fig. 8.** Example client program for the list ADT

### 2.4 Loops

Figure 7 shows function *dispose*, which takes a list of arbitrary length. It first frees all proper nodes, then it removes the remaining empty *lseg* assertion, then it frees the sentinel node, and finally it frees the **struct** *llist* object itself. A loop invariant must be provided for each loop. Execution of a loop proceeds by first consuming the loop invariant, assigning fresh symbols to the locals modified by the loop body, and producing the loop invariant again. Then execution proceeds along two branches: in one branch, the loop condition is produced, then the loop body is executed, and finally the loop invariant is consumed. If any chunks remain, this is considered a leak error. In the other branch, first the negation of the loop condition is produced, and then execution proceeds after the loop statement.

Figure 8 wraps up the example by showing an example client program for the list ADT. This program verifies; it follows that all assert statements succeed and no memory is leaked.

Notice that the SMT solver successfully evaluates the *add* fixpoint function applications.

## 3 Performance

The time complexity of verification is unbounded in theory. Specifically, since recursive pure functions of arbitrary time complexity may be defined, there is no bound on the time complexity of SMT queries. Furthermore, the approach, as currently implemented, does not perform joining of symbolic execution paths after conditional constructs; therefore, the number of symbolic execution steps is exponential in the number of such constructs. However, since no significant search is performed implicitly by the verifier or the SMT solver, performance is very good in practice.

The table below shows indicative verification times for a few example programs.

| program | total # lines | # annotation lines | time taken (seconds) |
|---|---|---|---|
| chat server | 242 | 114 | 0.08 |
| linked list and iterator | 332 | 194 | 0.09 |
| composite | 345 | 263 | 0.09 |
| JavaCard applet | 340 | 95 | 0.51 |
| GameServer | 383 | 148 | 0.23 |

## 4 Related work

Reynolds [17] introduced separation logic. Smallfoot [3] is a tool that performs symbolic execution using separation logic. This technique has been extended for greater automation [19], for termination proofs [4, 6], for fine-grained concurrency [5], for lock-based concurreny [11], and for Java [8, 12]. Unlike VeriFast, all of these tools attempt to infer loop invariants automatically.

Alternative specification and verification approaches, based on generation of verification conditions instead of symbolic execution, include VCC [7], Caduceus [9], ESC-Java [10], KeY [2], Jahob [20], regional logic [1], and approaches based on dynamic frames [13] including VeriCool [18], Dafny [14], and Chalice [15].

## 5 Conclusion

We presented an approach for specification and verification of imperative programs, that combines very good and predictable verification performance with powerful proofs written conveniently as part of the program. We are currently working to increase the degree of automation while preserving these strengths.

## Bibliography

[1] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008.

[2] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.
[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.
[4] James Brotheston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL*, 2008.
[5] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, 2007.
[6] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In *POPL*, 2007.
[7] Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE*, 2009.
[8] Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.
[9] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, 2007.
[10] Cormac Flanagan, K. Rustan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, 2002.
[11] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.
[12] Christian Haack and Clément Hurlin. Separation logic contracts for a Java-like language with fork/join. In *AMAST*, 2008.
[13] Ioannis T. Kassios. Dynamic frames: support for framing, dependencies and sharing without restrictions. In *FM*, 2006.
[14] K. Rustan M. Leino. Specification and verification of object-oriented software: Marktoberdorf international summer school 2008 pre-lecture notes, 2008.
[15] K. Rustan M. Leino, Peter Müller, and Jan Smans. *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*, chapter Verification of concurrent programs with Chalice. Springer, 2009.
[16] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.
[17] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.
[18] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *Workshop on Formal Techniques for Java-like Programs*, 2008.
[19] Hongseok Yang, Oukseh Lee, Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.
[20] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.